ELSEVIER

Computers and Electronics in Agriculture
27 (2000) 107–125

Computers and electronics in agriculture

www.elsevier.com/locate/compag

AppBuilder for DSSTools: an application development environment for developing decision support systems in Prolog

Geneho Kim [a], Donald Nute [a,*], H. Michael Rauscher [b], David L. Loftis [b]

[a] *Artificial Intelligence Center, Room 111 Boyd GSRC, The University of Georgia, Athens, GA 30602, USA*

[b] *Bent Creek Experimental Forest, USDA Forest Service, Asheville, NC 28801, USA*

Abstract

A programming environment for developing complex decision support systems (DSSs) should support rapid prototyping and modular design, feature a flexible knowledge representation scheme and sound inference mechanisms, provide project management, and be domain-independent. We have previously developed DSSTools (Decision Support System Tools), a reusable, domain-independent, and open-ended toolkit for developing DSSs in Prolog. DSSTools provides modular design, a flexible knowledge representation scheme, and sound inference mechanisms to support development of any knowledge based system components of a DSS. It also provides tools for building the DSS interface and for integrating other non-Prolog components of a DSS such as simulation models, databases, or geographical information system, into a multi-component DSS. DSSTools does not provide project management, and its complex syntax makes rapid prototyping difficult. AppBuilder for DSSTools is a GUI-based application development environment for developing DSSs in DSSTools that supports rapid prototyping and project management. AppBuilder's easy-to-use dialogues for managing and building knowledge based and top-level control components of a DSS free developers from having to memorize complex syntax and reduce development time without sacrificing the flexibility of the underlying toolkit. AppBuilder has been used to develop the Regeneration DSS, a system for predicting the regeneration of southern Appalachian hardwoods. AppBuilder is an application development environment for both prototyping and developing a complete DSS.

* Corresponding author.

0168-1699/00/$ - see front matter
PII: S0168-1699(00)00101-0

Keywords: Decision support systems; Knowledge based systems; Programming environments; Prolog

1. Introduction

A decision support system (DSS) is an interactive and flexible system that facilitates improved decision making, by integrating insights of decision makers as part of the decision making process (Schmoldt and Rauscher, 1996). The goal of a DSS is to amplify the power of decision makers without usurping their right to use human judgment and make choices. A DSS attempts to bring together the intellectual flexibility and imagination of humans with the speed, accuracy, and tirelessness of the computer. Rauscher (1999) provides a comprehensive review of DSS for ecosystem management in the US.

Traditional software engineering is a well-developed process for systematically designing a software product for well-behaved problems — problems that have known algorithmic solutions. Most DSSs in natural resource management are not well-behaved problems. Ambiguities, conflicts, internal inconsistencies, lack of organized solution strategies, institutional shock and confusion, and lack of scientific understanding all contribute to make natural resource management problems extremely complex. Designing DSSs for such problems can seldom be approached using classical software engineering methods.

The rapid prototyping approach to software engineering was developed for domains that have the above characteristics, i.e. when the requirements for the system or the decision making process cannot be fully known in advance (Pressman, 1992). Using prototyping methodology, an initial system that focuses on user interaction is quickly developed and enhanced based on evaluation by the user. This process of refinement is repeated until the resulting system performs satisfactorily (Schmoldt and Rauscher, 1996).

A programming environment that supports rapid prototyping of decision support systems for complex domains is needed. Such a programming environment should feature:

- short development and learning time;
- flexible knowledge representation schemes and sound reasoning methodologies;
- modular design allowing incremental refinement by parts;
- support for project management;
- capability to incorporate existing legacy programs and hypertext; and
- a domain-independent environment for developing domain-specific systems.

Short development time and a flat learning curve are the essence of a rapid prototyping environment. A DSS that preserves trustworthy decision-making throughout several modifications should employ a flexible knowledge representation scheme that can handle possible changes in the structure of the data. Inference mechanisms based on sound theoretical principles are also essential. Modular design and development is important because some stages of system improvement involve only parts of the system. Without project management, an inordinate

amount of time can be spent managing various files and components in a project. This is also important for software maintenance, particularly as the lifetime of software systems increases. Many decision support systems need to work with existing legacy software such as simulation models and geographic information systems, and with existing hypertext documents such as Windows help files. Therefore, the programming environment must provide tools for building interfaces with these systems and documents. Unlike expert system shells whose programs can be executed only within the shells, the desired programming environment must be able to generate domain-specific multi-component applications for a wide range of domains. To emphasize the application generation capability, we call such a programming environment an application development environment (ADE).

AppBuilder for DSSTools (AppBuilder for short) is a GUI-based ADE for developing the top-level control structure and knowledge-based systems component of decision support systems in any domain. AppBuilder is built on top of DSSTools, a Prolog toolkit for DSS development (Nute et al. 1995; Kim et al., 2000). AppBuilder facilitates rapid prototyping of decision support systems by providing easy-to-use dialogues for managing and building knowledge-based components of a system and integrating these with other components written in Prolog or some other programming language. Using AppBuilder, developers can reduce development time and eliminate syntax errors without sacrificing the flexibility of the underlying toolkit, DSSTools. In this paper, we briefly introduce DSSTools and its limitations, review the relevant literature, and then describe the AppBuilder ADE. This paper does not discuss tools in AppBuilder in detail. A detailed description of AppBuilder tools can be found in Kim (1999).

2. DSSTools and its limitations

DSSTools is a toolkit based on a blackboard architecture for developing decision support systems with a major knowledge-based component in Prolog. More information about DSSTools can be found in Nute et al. (1995) and Kim et al. (2000). For detailed discussion of the Prolog programming language, in which DSSTools is written, consult Covington et al. (1997). Developers can take advantage of a suite of tools available in DSSTools by using it as it is or modifying it to suit their needs, thus reducing development time over developing a system from scratch in Prolog. The modifiability of DSSTools provides a flexibility missing from the typical commercial expert system or other DSS development tool. DSSTools is open-ended, meaning that the current suite of tools can be continuously enhanced and new tools can be added.

DSSTools is designed to support development of multi-component DSSs that combine knowledge based systems and other kinds of DSS components using a blackboard architecture. One or more semi-autonomous agents are developed that have access to the information on the blackboard and can post new information to the blackboard. These agents are called domain control modules (DCMs). A single DCM might provide an interface with the user, implement a knowledge-based

system, or manage another kind of DSS component developed in Prolog or some other programming language. DCMs can be independently developed, executed, and tested. This modular structure allows incremental refinement by parts — the parts being DCMs. DCMs do not communicate with each other directly; they communicate through shared data on the blackboard. DSSTools provides reusable codes to maintain the blackboard and to provide the DCMs with the opportunity to interact with the blackboard.

DSSTools provides a rich, flexible fact structure that includes an attribute-object-value (AOV) triple, an inference index that may be used to represent degree of confidence, and the source of a fact. Because there are no restrictions on the values and the uses of the inference indexes and sources, they are available to support different inference mechanisms such as fuzzy logic or reasoning with confidence factors. DSSTools provides backward-chaining and forward-chaining inference engines, for use with backward-chaining rules and forward-chaining rules, respectively. These rule types and inference engines form a sound and widely used standard reasoning mechanism.

DSSTools also provides interface tools for gathering information from users or external programs. DSSTools supports six types of GUI dialog screens for handling different types of input. These dialog screens are called query screens because they provide information that the system needs by querying the user. DSSTools also features tools for interfacing with foreign components (simulations, spreadsheets, geographical information systems, etc.) and the Windows help system. A specific page in a Windows help file can be linked to a particular rule or a specific query screen. Using the DSSTools interface tools, a GUI dialog screen with a number of input fields can be generated with less than ten lines of code.

Although DSSTools provides many enhancements to the native Prolog programming environment, the toolkit still fails to provide all the features of an ADE described above. Developers must type all components of the DSSTools application by hand. Even for a moderately sized application, the amount of code one must manage is too large to handle without assistance of project management tools. To ensure that a DSSTools application works properly, we need to test components of the application. Both project management and easy component testing are missing in LPA Win-Prolog, the software environment for developing applications in DSSTools (LPA Win-Prolog, 1996). DSSTools users must possess at least a moderate understanding of Prolog programming and learn the complex syntax for each tool in the toolkit (Kim et al., 2000). This process takes time, for learning as well as software development, and does not protect the developer from making frequent syntax errors. Furthermore, to use the powerful user interface routines in DSSTools, one must master the complex syntax of each interface type. The deep structured syntax also makes modification of the interface difficult. AppBuilder was created as an ADE for DSSTools to mitigate some of the weaknesses inherent in the toolkit approach to developing DSSs.

3. Related work

Previously developed toolkits and programming tools provided insight for development of AppBuilder. Micro Interpreter for Knowledge Engineering (MIKE) (Eisenstadt and Brayshaw, 1990a,b), a knowledge engineering toolkit for building expert systems in Prolog using the frame knowledge representation structure, is very close to DSSTools in terms of functionality. One notable difference is that MIKE can be used with any Prolog programming environment whereas AppBuilder requires a particular Prolog programming environment, LPA Win-Prolog. Karsai (1995) introduces a configurable model-based visual programming environment (VPE) that can be configured for developing systems in various domains. Koseki et al. (1996) propose an architecture for a VPE for hybrid expert systems. Their visual environment pointed out the desirability of a visual programming tool for developing what we call domain control modules. Other systems we reviewed demonstrated that approaches we have taken are not unique to DSSTools and AppBuilder. Like DSSTools, THESEUS + +, an object-oriented toolkit for developing high level user interfaces for 2.5D graphics, separated problem solving knowledge from GUI-related code (Dingeldein and Lux, 1993). Like AppBuilder, Kim's QA Builder (Kim, 1996) features visual editors developed for a particular kind of knowledge representation scheme. Like AppBuilder, FIELD (Friendly Integrated Environment for Learning and Development) (Reiss, 1995) provides graphical interfaces for existing programming tools that enhance the functionality of the original tool. Since none of these systems meets our requirements for an ADE, we developed a new programming environment using DSSTools. We chose DSSTools over other toolkits, such as MIKE, because it has been used to develop a real-world application (Nute et al., 1995) and because it satisfies all of our requirements except short development time and project management support.

4. AppBuilder for DSSTools

AppBuilder is an ADE for DSSTools that supports short development time and project management. AppBuilder reduces development time by providing easy-to-use GUI tools to manage and develop components of a system and by freeing developers from having to memorize and learn complex syntax of DSSTools features.

Fig. 1 depicts the relationship between AppBuilder tools and DSSTools components. White boxes in the figure represent AppBuilder tools and gray boxes represent typical components of a DSS developed using AppBuilder. Gray boxes appearing to the left of the dotted vertical line are non-DSSTools parts of the system. These will typically include simulations, databases, geographic information systems, or other DSS components besides the knowledge-based components developed using AppBuilder. Each AppBuilder tool is linked to a DSSTools component related to that tool. The Project Window encloses all DSSTools components to emphasize that it is used to manage all DSSTools components.

AppBuilder provides two levels of project management functionality: project-level and component-level. Project-level functionality concerns manipulating projects, which includes creating a new project and renaming an existing project thus allowing developers to save different versions of a project in different locations. It also includes adding files to and removing files from projects. Component-level functionality involves manipulating components contained in a project. Examples of components are DCMs, knowledge bases, and user interface screens. The separation of these two levels of functionality enables developers to focus their attention either on the project-level as a whole or on the design and implementation of one component of an application at a time. In this way, whether to use the bottom-up or top-down software development strategies is left to the discretion of the developer.

Project-level functionality is provided by the Project Window (Fig. 1). The Project Window is essentially the command center for the DSS software developer. In AppBuilder, a project is a set of files that make up a DSSTools application. A DSSTools application must have one or more domain control module (DCM) files and exactly one main file that contains start-up code. Optionally, an application

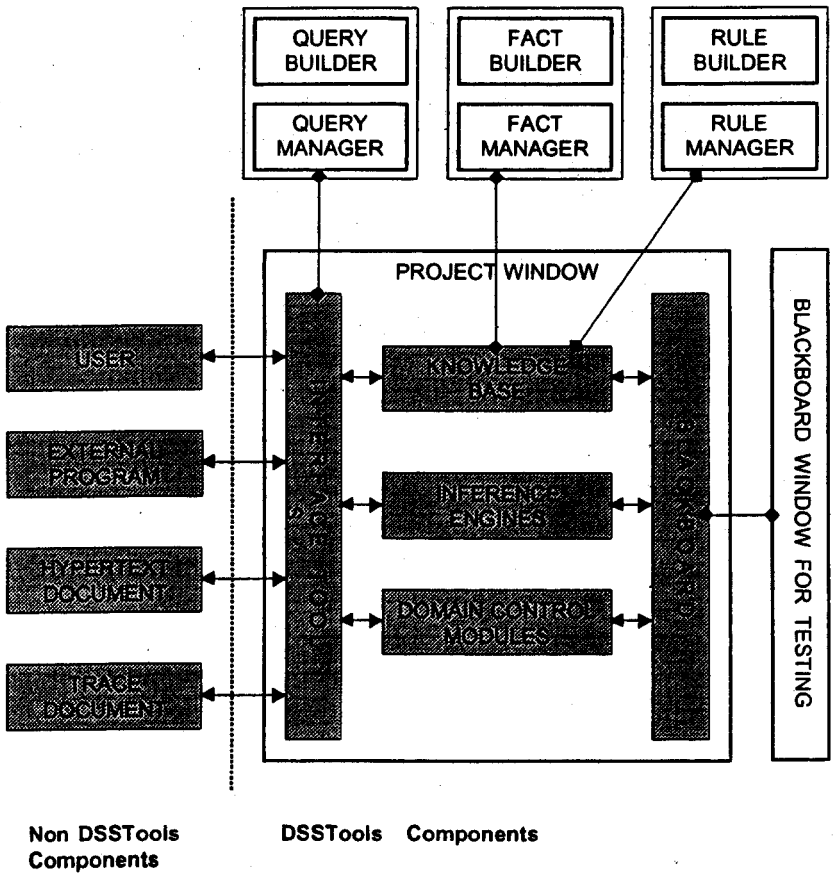

Fig. 1. AppBuilder Tools and DSSTools components.

may contain one or more knowledge base files, one or more domain utility (DUT) files, Windows help files, and other non-Prolog components that are not tracked by AppBuilder. Managing projects with less than ten files by hand without a GUI tool is not difficult. However, when the number of files in a project grows larger than that, the task of managing a project gets out of hand and AppBuilder becomes increasingly more valuable.

Another important project-level feature of AppBuilder is project verification. AppBuilder implements a strategy not just for syntactic error checking, but also for consistency. In consistent projects, no two rules or queries have the same name, all files actually exist, and a DCM that terminates the system exists. Rather than reinventing the wheel, AppBuilder uses the built-in syntax checking and debugging capabilities of LPA Win-Prolog.

Component-level functionality is provided by component managers and builders (Fig. 1). The component managers in AppBuilder are the Fact Manager, the Rule Manager, and the Query Manager. Component managers allow us to review, add, modify, or delete components in an application. For some of these functions the component managers call component builders, specialized dialog screens for creating and modifying system components. There is a Fact Builder, a Rule Builder, and a Query Builder. We will see screen images for several of the manager and builder dialogs when we walk through the development of a DSS using AppBuilder later in this paper. To reduce the learning time for the user, AppBuilder's managers have a uniform 'look-and-feel'. Builders, on the other hand, are more distinctive because they must deal with completely different types of components.

5. Designing the Regeneration DSS

An example will illustrate how AppBuilder can be used to develop a fully functioning knowledge-based DSS. To concentrate on the functionality of AppBuilder, this sample application does not include any non-DSSTools components. Loftis (1990) describes a conceptual model for predicting the amount and species composition of regeneration for southern Appalachian hardwood forests based on the initial floristic composition theory of forest ecology. This theory states that the individuals of species that form the dominant forest canopy 10 years after a heavy disturbance event come from the initial load of propagules present on the site prior to the disturbance. The Regeneration DSS requires as input the number, size, and species of regeneration propagules prior to a disturbance event. The output from the regeneration DSS is the number, size, and species of forest trees present on the site 10 years following the heavy disturbance event. AppBuilder was used to develop the regeneration DSS and Dr David Loftis, Principal Silviculturalist with the USDA Forest Service's Bent Creek Experimental Forest in Asheville, NC, performed the role of domain expert.

Since the input and output requirements for the Regeneration DSS were well known from the beginning, the major problem definition work concentrated on the identification of domain control modules, their functionality, and their

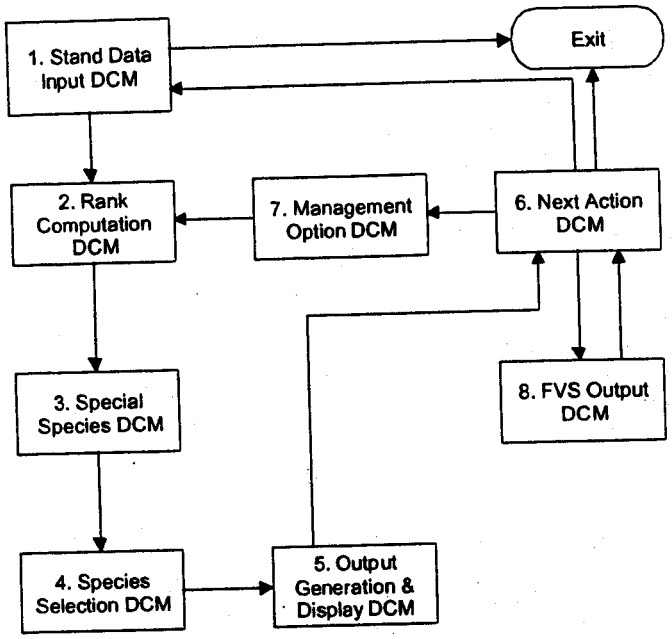

Fig. 2. Domain control modules for the Regeneration DSS.

interrelationships (Fig. 2). We identified eight separate functions that the DSS would perform, and a DCM was created for each of these functions. The Stand Data Input DCM runs first and allows the user to enter new stand data, read an existing stand data file, or exit the system. Once data is entered or read, the Rank Computation DCM computes the rank of each species. The rank of a species is a measure of its competitive advantage during regeneration. Species ranks are used to compute the likelihood that a certain species will survive after a heavy disturbance. The Special Species DCM asks the user to select species present in the stand that are exceptions to the general Regeneration logic. The Species Selection DCM selects the species that win the competition during regeneration and produce trees in the overstory. Once the Output Generation and Display DCM generates output in HTML format and calls the user's default internet browser to display it. The Next Action DCM presents the user with choices. Users can explore management options, such as removing trees of a certain size and species, or send the output of the system to a file properly formatted for the Forest Vegetation Simulator (FVS), a growth simulator developed by USDA Forest Service (Teck et al., 1996). These two options are handled by the Management Option DCM and the FVS Output DCM. Users can also exit or analyze another stand data set. A more detailed description of each DCM and its functionality will be presented in Loftis et al. (2000).

The Regeneration DSS does not include rules. Instead, the knowledge base contains a set of facts recording the ranks of different species. An algorithm implementing the decision process uses information about understory plants present on the stand prior to the disturbance event, about species present in the canopy

prior to the disturbance that produce stump sprouts, and about trees present in the surrounding area to predict which trees will survive as members of the eventual canopy. This algorithm was written in Prolog and incorporated into the Rank Computation DCM.

Next, each DCM was analyzed for user interface screens. We determined that we would need three query screens. One screen asks the user for a site index for the stand being analyzed. Another asks the user to select special species present in the stand. The third screen allows users to make a selection among choices that correspond to the Next Action DCM. The three query screens will be linked to a previously developed Windows help file that provides an organized synthesis of knowledge concerning the entire southern Appalachian regeneration domain (Rauscher and Host, 1990; Rauscher et al., 1997). The Regeneration DSS also needed a spreadsheet like input screen to enable users to enter stand data and a system to generate output as an HTML document. Since AppBuilder does not support such spreadsheets or generation of HTML documents, we built both the spreadsheet input screen and HTML document generation routines without using AppBuilder. Fig. 3 shows the spreadsheet query screen and Fig. 4 shows an example of output from the HTML report generator displayed using Netscape. We plan to add the HTML report generator and the spreadsheet query screen to our

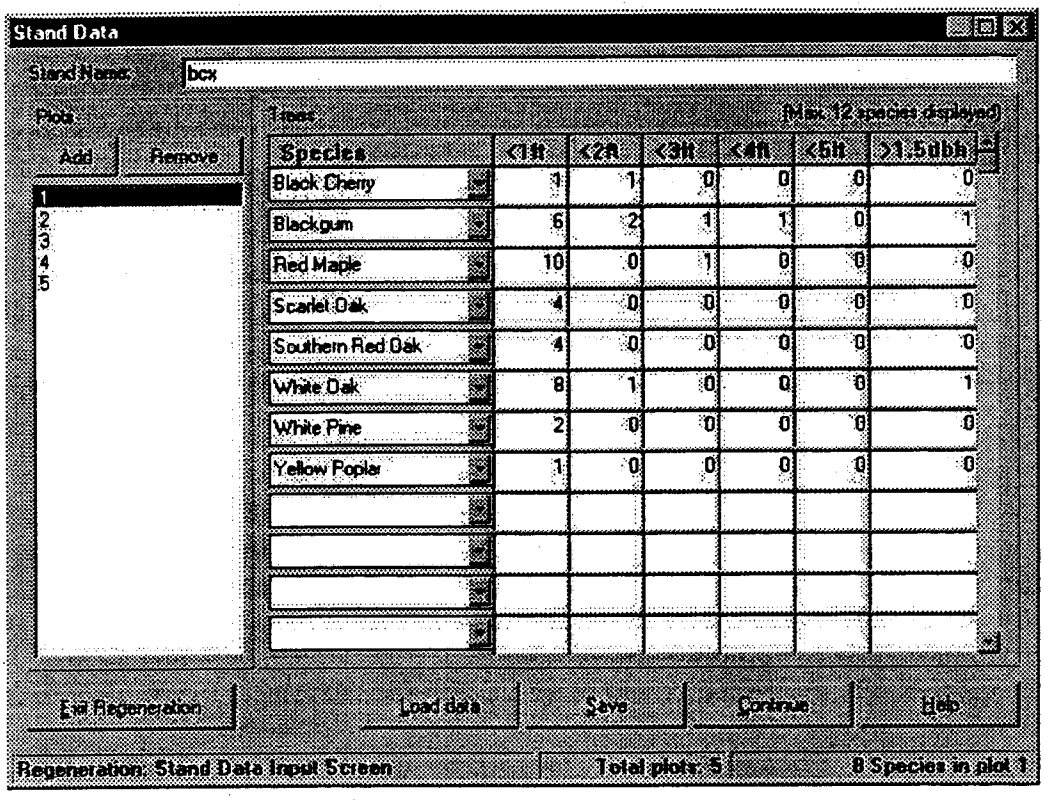

Fig. 3. Stand data spreadsheet query screen.

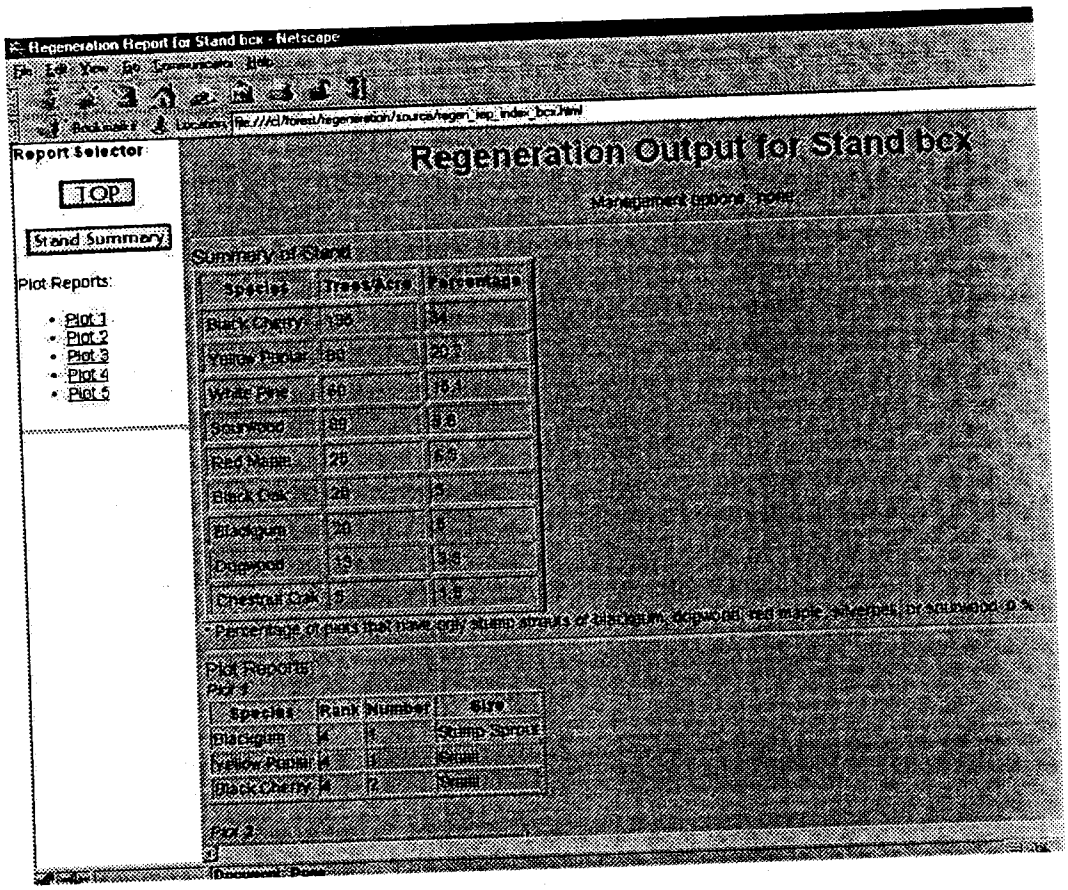

Fig. 4. An HTML report from the Regeneration DSS.

next version of DSSTools and AppBuilder; so we developed them to work independently of any code specific to the Regeneration DSS.

6. Implementing the Regeneration DSS using AppBuilder

Having developed the application design, we were ready to implement the first prototype. Fig. 5 shows the LPA Win-Prolog console window after AppBuilder is loaded. Notice that an AppBuilder menu item has been added to the menu bar and that all the DSSTools and AppBuilder components have been consulted. The first step is to create a new project. When we click on New DSST Project in the AppBuilder menu, we must provide a name for the new project and choose the directory in which project files will be saved. Naming the project Regen, the Project Window is displayed (Fig. 6) showing the name of the main project file (regen.pl) and an initially unnamed knowledge base file for the project (untitled.kb). Clicking on regen.pl in the Project Window, then clicking the Edit button, we are presented with a text window displaying the newly created main project file (Fig. 7). Notice

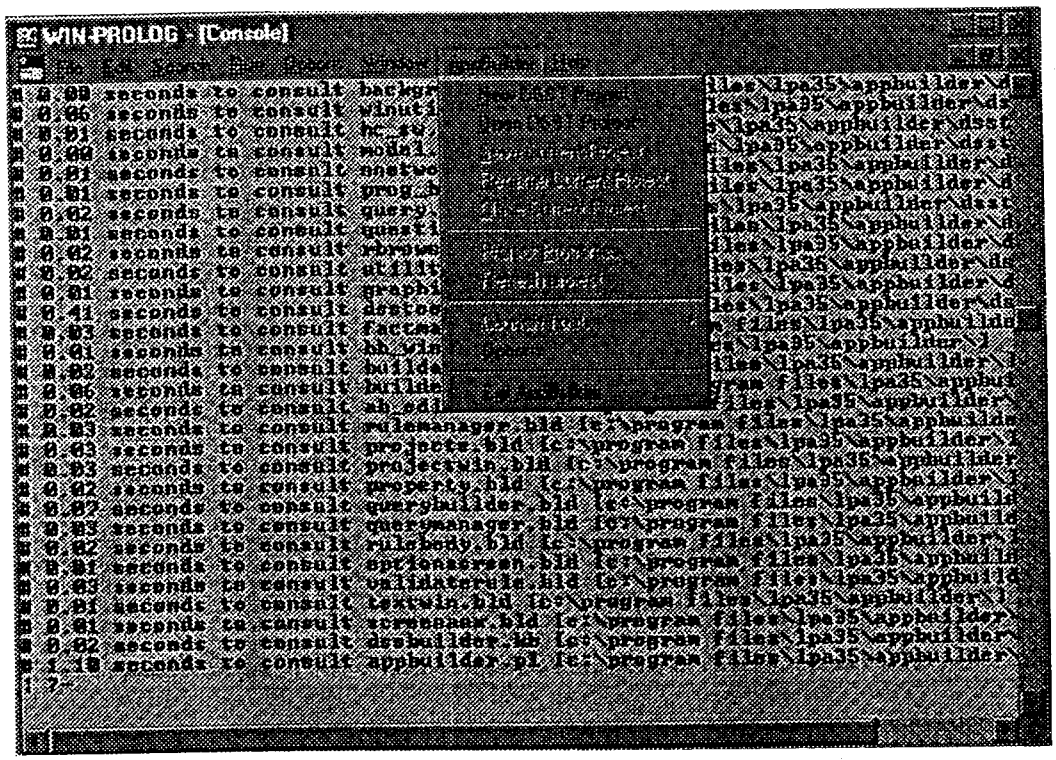

Fig. 5. The LPA Win-Prolog console with the AppBuilder Menu added.

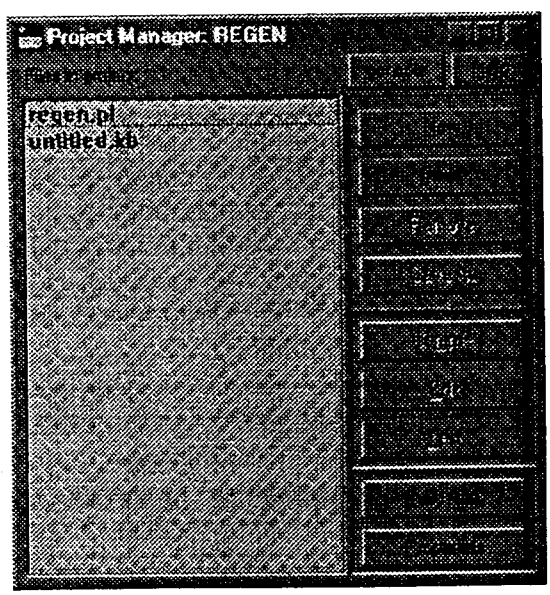

Fig. 6. The Project Window with the newly-created Regeneration project.

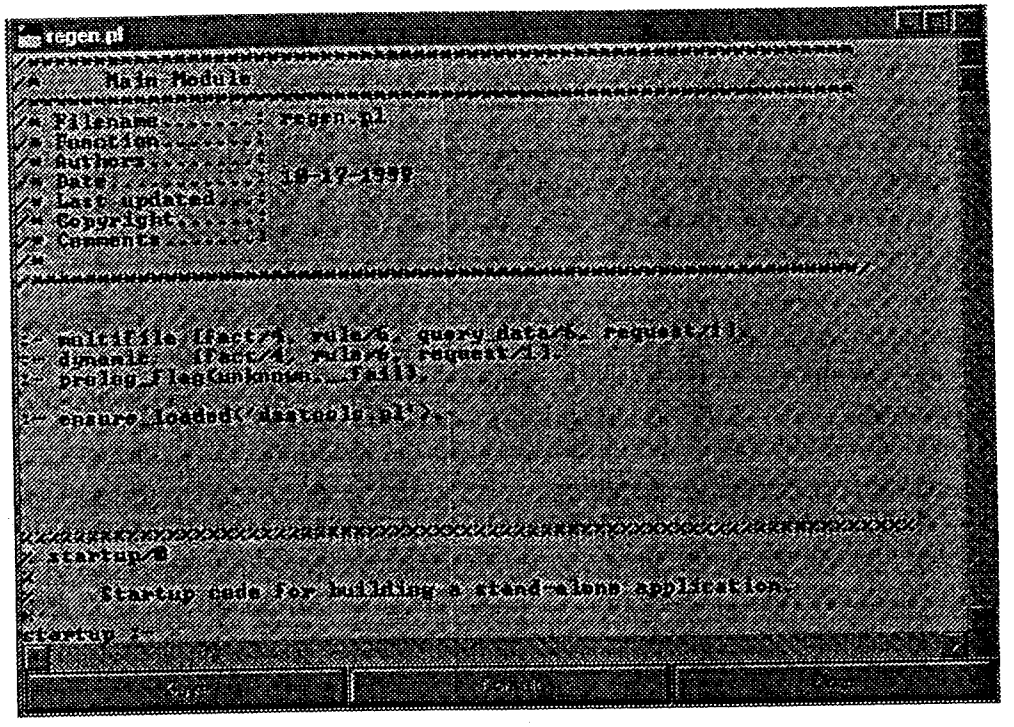

Fig. 7. The main project file for the Regeneration DSS.

that regen.pl already contains a header where important information about the file can be placed. If we click the Properties button in the Project Window dialog, we can provide author names and copyright information. As each new file is added to the project, a header is automatically created for the file and the author names and copyright information are placed in the header. Notice also that regen.pl contains some initial code required in any DSSTools main file, although most of this initial code is not visible in this screen image.

After the project is created, we name the initial knowledge base (in this case, Regen1.kb) and add the DCMs, DSSTools domain utility files (with extension .dut), and Windows Help files to be included in the project. A domain utility file contains domain specific codes used by several different DCMs. The user may add these files to the project and develop them one at a time. However, we have found it helpful to create every DCM file that we think we will need and add code to the DCM that simply looks for a request for the DCM to act. When such a request is found on the blackboard, the system displays a message telling the user the DCM is active. We may also include code that puts requests on the blackboard for one or more additional DCMs to act. This allows us to test the flow of control between the DCMs before we begin serious development of any single DCM.

The Project Window in Fig. 8 shows a complete list of all the files in the Regeneration DCM. We can click on any DCM file in the list and edit it or test it. We can also test the entire application by clicking on the main project file (the one with the .pl extension) and then clicking the Test button. Before testing either an

individual DCM or an entire application, a Blackboard Editor dialog (Fig. 9) appears. This allows the user to type in a set of test facts to appear on the blackboard, to save a set of test facts once entered, or to load a set of previously saved test facts to the blackboard before starting the DCM or application. This facilitates modular testing of system components as they are developed.

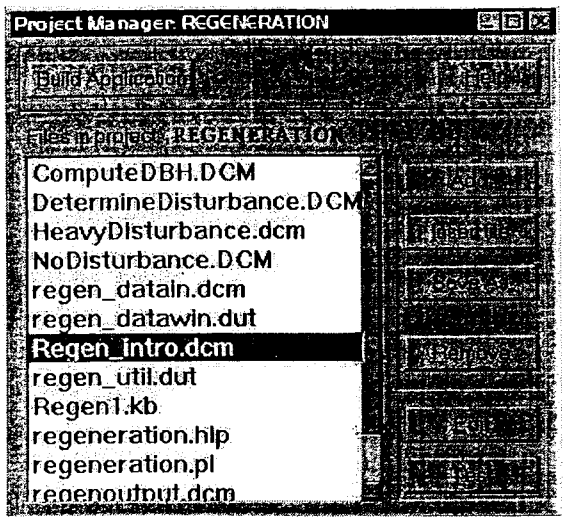

Fig. 8. The Project Window with the complete Regeneration project.

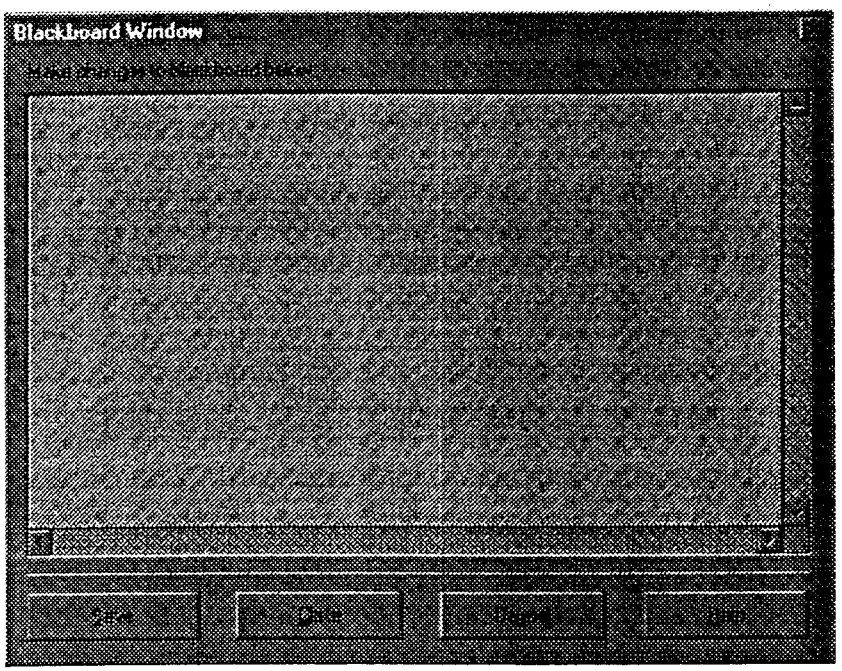

Fig. 9. The Blackboard Editor.

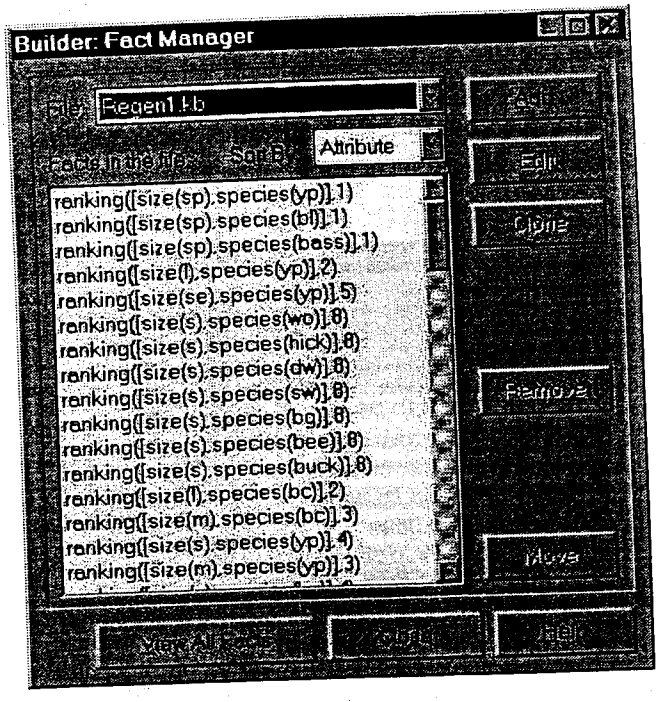

Fig. 10. The Fact Manager displaying facts in the Regeneration knowledge base.

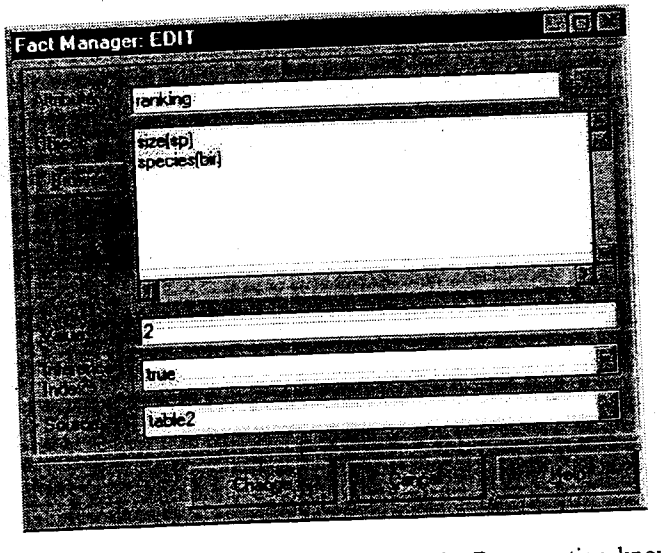

Fig. 11. The Fact Builder during editing of a fact in the Regeneration knowledge base.

The next step was to use the Fact Manager and Builder tools to write facts to the Regeneration project knowledge-base file. Figs. 10 and 11 show the Fact Manager and Fact Builder as they were used to add facts to the Regeneration knowledge base. This set of facts is used to determine the likelihood that a certain species will survive after a heavy disturbance. The majority of the facts added were similar to

each other, requiring only one or two changes. Thus, we used the 'clone' function of the Fact Manager. With the cloning function, we were able to add more than 90 facts in less than half an hour. Without AppBuilder, it would have taken us considerably longer including the time to debug any potential syntax and structural errors.

Next, a test input data set was created using the Fact Manager and Fact Builder. We created a new KB file in the project and added facts to the KB file. After facts were built, rules would normally be built, using the Rule Manager and the Rule Builder. Since Regeneration uses no rules, this step was skipped. The three required query screens were created and tested using the Query Manager and the Query Builder. We linked each query screen to a specific page in the Regeneration help file that contains detailed explanation for each screen. Fig. 12 shows one of the query screens, the Special Species query screen.

After developing the necessary query screens, we returned to development of the eight DCMs. We replaced the original temporary displays in the Stand Data Input DCM, Special Species DCM, and the Next Action DCM with the DSSTools query screens we had developed. We also developed the spreadsheet query screen that is called by the Stand Data Input DCM. The algorithms for the Rank Computation DCM and the Species Selection DCM were added to these DCMs, and the query screens were removed from all DCMs that would not communicate directly with

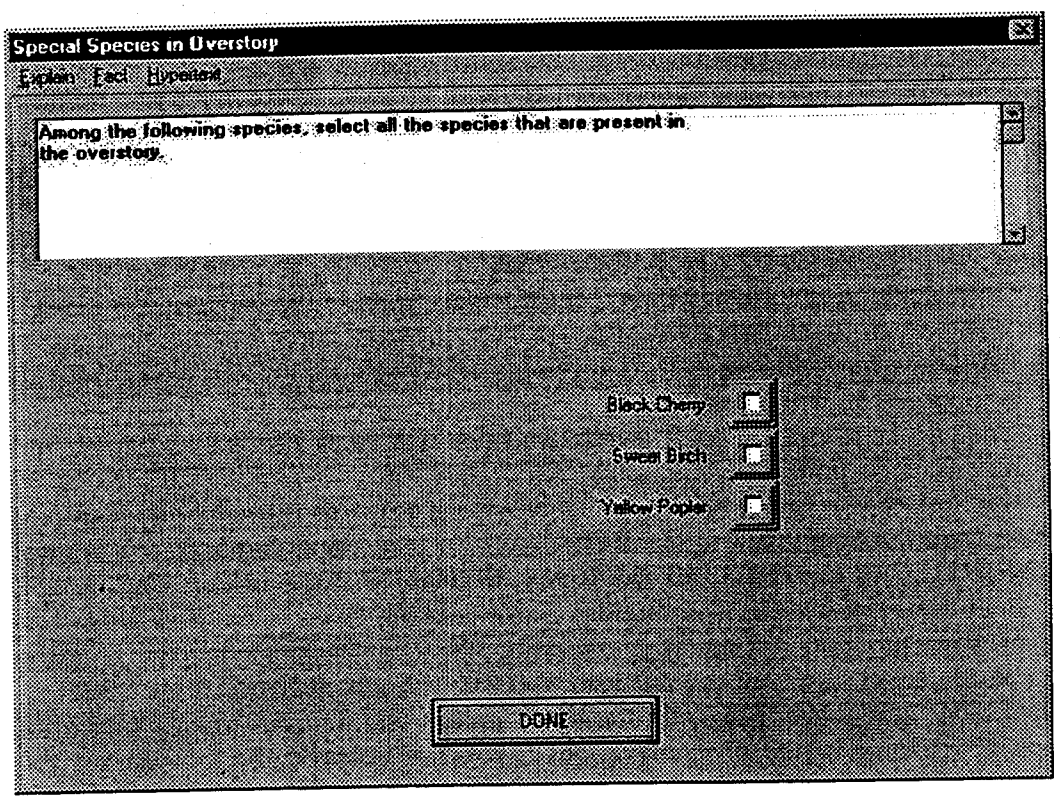

Fig. 12. The Special Species query screen.

the user in the final system. We then developed the routines that generate output as an HTML document. All development of the actual DCMs was done in the corresponding file text windows like the one in Fig. 6. Since almost anything can be included in a DCM file, there is no DCM builder dialog. The entire system was tested from the Project window, again using the Blackboard Window.

As the final step in the development process, we generated a stand-alone application by clicking the Build Application button from the Project Window. AppBuilder automatically validated the project by conducting various internal consistency checks before generating the stand-alone application. Because building the system using the builders and managers eliminates syntax errors and because AppBuilder does not allow any inconsistent addition or modification of components, the Regeneration project was consistent and free of syntactic errors.

7. Discussion

Although no formal, independent evaluation has yet been performed, AppBuilder was evaluated based on our experience of building applications with it. The evaluation criteria for AppBuilder address two aspects of the system: functionality and capacity. The evaluation criteria regarding the functionality of the system are identical to the requirements of an ADE as discussed previously. AppBuilder satisfies all these requirements. AppBuilder provides a set of dialogues to build and manage various components of a DSSTools application. The Project window provides project management capability, and its testing facility eases modular design and testing. AppBuilder does not restrict domains for which it can generate domain-specific applications. Although it does not provide an explicit version control function, the ability to rename projects can be used to handle different prototypes and versions. Other requirements are satisfied simply because the underlying toolkit, DSSTools, satisfies them.

The criteria used to evaluate AppBuilder that address the capacity of an ADE with respect to its underlying toolkit are:

- Efficiency: how efficient is it to develop a DSSTools application using the ADE as opposed to without it?
- Completeness: can an ADE generate all possible applications that can be built just using its underlying toolkit?

AppBuilder is efficient. The amount of time it takes to learn AppBuilder is no more than the time taken to learn DSSTools. In fact, using AppBuilder is a quick way to learn to use DSSTools. Moreover, developing Regeneration DSS in AppBuilder demonstrated that AppBuilder clearly reduces development time in both the implementation and testing stages of an application development process. In the implementation stage, we did not have to learn the complete syntax of DSSTools. This alone saves time. Furthermore, if we build a query screen using a text editor and try running it, the probability that it will run properly the first time is very small due to the complexity of the syntax. The query may not contain syntax errors but be improperly structured. Then the query will not run although there are no

syntax errors. Finding bugs caused by improperly structured DSSTools components that are legitimate Prolog code normally consumes significant amounts of resources because such errors cannot be detected at compile time and because Prolog cannot find them. If we use the Query builder, however, a query screen will run the first time almost always because AppBuilder ensures that the query is error-free. This is equally true for facts and rules. Hence, AppBuilder saves significant resources that might otherwise be used in finding and correcting errors.

The resources spent in the testing stage were also reduced. Most of the time, we want to verify that a DCM or an entire application works with a number of different configurations of the blackboard and not just with one. To test a DCM without AppBuilder, we first have to modify the contents of the blackboard either by writing a file that contains the desired contents of the blackboard or by executing a series of blackboard modification functions DSSTools provides. Then we should make sure that the modifications we make are error-free. If they contain errors, the DCM we test will not run properly. In that case, it is not easy to determine whether the bug is in the DCM or in the modifications we made to the blackboard. We must repeat these cumbersome steps for each configuration we want to test the DCM against. In AppBuilder, we simply click the Test button after selecting the main file in the project. Then using the Blackboard Window, we can easily modify the contents of the blackboard. We can even save the changes we make for later use. Since the Blackboard Window does not allow improperly structured modifications, we can rest assured that if the DCM does not run properly, it is the DCM that contains a bug and not the blackboard modification we made. We can even create a knowledge base that contains a desired configuration using AppBuilder's component managers and builders. To do so, we simply create a new knowledge base file and insert it into the project. After we have tested the desired DCM, we can remove the file from the project without affecting other parts of the project. This is what we did with the Regeneration DSS. We can test the entire application just as we would test individual DCMs. Testing a query screen is also easier. Once we create a new query screen, we simply click the Test button to run the query. Without AppBuilder, we would have to call a specific DSSTools routine with proper arguments to test a query screen.

The question whether AppBuilder is complete with respect to DSSTools cannot be answered with a simple yes or no. AppBuilder allows us to manage all the components of a DSSTools application and it provides tools to build major components of a DSSTools application: facts, rules, and queries. But it does not provide the same kind of facility for constructing DCMs. Unlike facts, tools, and queries, a DCM does not consist of some definite set of parts that never alter. We can include anything in a DCM. In the case of the Regeneration DSS, we included special algorithms that capture the logic of choosing winners in a regeneration scenario and we included a custom spreadsheet interface screen. We cannot keep the open-ended architecture of a toolkit like DSS and also anticipate every domain specific task a developer might need to incorporate into an application. So the best we can do is to build the headers and first bits of code for each DCM and provide text editing screens where the developer can write the rest of the DCM. The

strength of a toolkit is flexibility, and the strength of an ADE is that it makes it easier for the developer to do routine tasks. These two goals are not entirely compatible since flexibility means making it possible to do what is not routine. The combination of DSSTools and AppBuilder represent a compromise.

Developers should remember that AppBuilder works only with LPA Win-Prolog. We also emphasize that AppBuilder inherits all the limitations of DSSTools. While DSSTools is designed to support development of knowledge based systems and to simplify integration of other kinds of DSS tools with a knowledge based system, neither DSSTools nor AppBuilder provides the tools to develop those additional DSS components such as simulations, databases, or geographical information systems.

To enhance project management functionality, we plan to add a printing function that produces a hard copy of a project and of component files in a project. This will help developers document their code. In addition, we plan to develop a visual tool for designing DCM execution flow. This would enable developers to visualize system behavior and to design DCMs more effectively. As we mentioned, DSSTools is open-ended and new tools will be added to it. Hence, AppBuilder will also continue to enhance its features to help manage new tools added to DSSTools.

Acknowledgements

The development of AppBuilder for DSSTools was funded by the USDA Forest Service, Bent Creek Experimental Forest, Asheville, NC through a cooperative research agreement with the Artificial Intelligence Center of the University of Georgia.

References

Covington, M.A., Nute, D., Vellino, A., 1997. Prolog Programming in Depth. Prentice-Hall, Upper Saddle River, NJ.

Dingeldein, D., Lux, G., 1993. THESEUS + +: a high level user interface toolkit for graphical applications. Comput. Graphics 17 (2), 147–154.

Eisenstadt, M., Brayshaw, M., 1990a. A knowledge engineering toolkit — your own knowledge engineering toolkit for building expert systems. BYTE 15 (10), 268–282.

Eisenstadt, M., Brayshaw, M., 1990b. A knowledge engineering toolkit, part 2 — your own knowledge engineering toolkit for building expert systems. BYTE 15 (12), 346–370.

Karsai, G., 1995. A configurable visual programming environment: a tool for domain-specific programming. IEEE Comput. 28 (3), 36–44.

Kim, M.Y., 1996. QA Builder: a visual toolkit for building multimedia knowledge-based systems with immediate feedback. In: Proceedings of the International Conference on Multimedia Computing and Systems, Los Alamitos, California. IEEE, pp. 269–273.

Kim, G., 1999. AppBuilder for DSSTools. MS Thesis, Artificial Intelligence Center, University of Georgia, Athens, GA.

Kim, G., Nute, D., Rauscher, H.M., Maier, F., 2000. DSSTools 2000: A Toolkit for the Development of Decision Support Systems in Prolog. USDA Forest Service, Southern Research Station, Asheville, NC, GTR-SRS-XXX in press.

Koseki, Y., Tanaka, M., Maeda, Y., Koike, Y., 1996. Visual Programming Environment for Hybrid Expert Systems. Expert Syst. Applications 10 (3–4), 481–486.

Loftis, D.L., 1990. Regeneration of southern hardwoods: some ecological concepts. In: Proceedings of the National Silvicultural Workshop, July 10–13, 1989, Petersburg, AL. US Department of Agriculture, Forest Service, Washington, DC, pp. 139–143.

Loftis, D.L., Rauscher, H.M., Kim, G., Nute, D.E., Rushton, N., 2000. Regeneration: A Decision Support System for Predicting Southern Appalachian Hardwood Regeneration. USDA Forest Service, Southern Research Station, Asheville, NC, GTR-SRS-XXX (in preparation).

LPA Win-Prolog, 1996. Version 3.5. Computer software. Logic Programming Associates.

Nute, D.E., Rauscher, H.M., Zhu, G., Chang, Y., Host, G.E., 1995. A toolkit approach to developing forest management advisory system in Prolog. AI Applications 9 (3), 39–58.

Pressman, R.S., 1992. Software Engineering: A Practitioner's Approach, third ed. McGraw-Hill, New York.

Rauscher, H.M., 1999. Ecosystem management decision support for federal forests in the US: a review. Forest Ecol. Manag. 114, 173–197.

Rauscher, H.M., Host, G.E., 1990. Hypertext and AI: a complementary combination for knowledge management. AI Applications 4 (3), 56–61.

Rauscher, H.M., Loftis, D.L., McGee, C.E., Worth, C.V., 1997. Oak regeneration: a knowledge synthesis. Compiler 15 (1), 51–52 insert, three disks, 3.8 megabytes; 748 chunks, 742 links (electronic).

Reiss, S.P., 1995. The Field Programming Environment: A Friendly Integrated Environment for Learning and Development. Kluwer, Boston, MA.

Schmoldt, D.L., Rauscher, H.M., 1996. Building Knowledge-Based Systems for Natural Resource Management. Chapman and Hall, New York.

Teck, R., Moer, M., Eav, B., 1996. Forecasting ecosystems with the forest vegetation simulator. J. For. 94 (12), 7–10.